PaMiraXT: Parallel SAT Solving with Threads and Message Passing

Tobias Schubert schubert@informatik.uni-freiburg.de
Matthew Lewis lewis@informatik.uni-freiburg.de
Bernd Becker becker@informatik.uni-freiburg.de
Institute of Computer Science,
Faculty of Engineering,
Albert-Ludwigs-University Freiburg

Abstract

This article describes PaMiraXT, a powerful parallel SAT algorithm. PaMiraXT follows a master/client model based on message passing, making it suitable for any kind of workstation cluster. For the clients, MiraXT is used, which itself is thread-based parallel solver designed to take advantage of current and future shared memory multiprocessor systems. We highlight design and implementation details that allow the threads/clients to run and cooperate efficiently. Experimental results show that MiraXT compares well to other state-of-the-art SAT algorithms. In single-threaded mode, it outperforms MiniSat2, PicoSAT 535, and RSat 2.01, while in multi-threaded mode, MiraXT provides cutting edge performance, as it solves significantly more instances within the given time limit. A case study, using three copies of MiraXT with a total of 8 threads as clients, underlines the potential of PaMiraXT, resulting in a speedup of 5.62 on the industrial benchmarks of the 2007 SAT competition.

Keywords: *parallel SAT solving, threads, message passing*

Submitted November 2008; revised May 2009; published June 2009

1. Introduction

Todays sequential SAT solvers are considerably more advanced than the original Davis-Putnam algorithm [7, 8] introduced more than 45 years ago. Many performance enhancements have been of more algorithmic nature, such as non-chronological backtracking [21], conflict-driven learning [31], novel decision heuristics [13, 23], restarts [16], and powerful preprocessing techniques [2, 9]. On the other hand, the implementations of these methods have been improved by considering the underlying hardware environment that was running the SAT solver. For example, zChaff introduced the concept of watched literals, that effectively uses the cache memory of modern CPUs [32].

With all these features, modern SAT algorithms are now able to tackle Boolean satisfiability problems with hundreds of thousands of variables (or sometimes even more). However, the last few years have shown two trends: first, the performance gains achieved by the SAT community in the field of sequential SAT solving are getting marginal, as it becomes harder and harder to optimize current algorithms even more. As a result, many of the challenging problem instances are still unsolvable by any sequential SAT solver within a reasonable time frame. Second, many chip manufacturers have turned from single-core to multi-core designs as a way to increase the performance of their processors as gains in processor frequency have stalled. Companies such as AMD, IBM, Intel, and SUN now produce CPUs that contain four or more cores per processor. Putting these two observations together, a new generation of parallel SAT solvers is needed to utilize the enormous potential of multi-core processors and multi-CPU workstations and by this to reach the next performance level of modern SAT solving.

In this article we present the parallel SAT solver PaMiraXT that has been optimized to run efficiently on many different hardware platforms. The solver consists of two layers of parallelism. The first level is based on MiraXT, a thread-based implementation optimized for shared memory multiprocessor systems (multi-core and/or multi-CPU). Each thread of MiraXT consists of a complete zChaff-like SAT procedure with all the features mentioned earlier being integrated into the solver. The key feature of MiraXT is that only one global clause database, the so-called *Shared Clause Database*, is maintained, where all clauses of the original CNF formula and all conflict clauses deduced by the threads get stored. The main advantage of this design is the fact that each thread is able to check all available clauses. Each thread can consider its current status when deciding which conflict clauses might be helpful to guide the current search process. Finally, MiraXT contains no controlling master process, but a *Master Control Object* to allow the threads to communicate with each other. All communication is done in a passive way by storing signals and messages that get polled by the threads occasionally to see if there are any new messages. Our performance evaluation on a workstation equipped with two *Dual-Core AMD Opteron 280* processors shows, that MiraXT provides almost linear speedup on hard instances in the multi-threaded mode, while being competitive to other state-of-the-art sequential SAT algorithms in the single-threaded mode. The last point clearly indicates that the algorithmic overhead to handle multiple threads running in parallel can be quite low compared to a sequential SAT solver, which has all its data structures optimized for just one SAT solving routine.

To be able to run our solver not only on shared memory multiprocessor systems, but also on classical workstation clusters, we extended it with a second layer of parallelism, resulting in the entire parallel SAT algorithm PaMiraXT. The overall design follows a master/client model, where copies of MiraXT are run on different workstations of a particular cluster (acting as clients). For communication purpose we added a separate master process, responsible for exchanging status signals, unevaluated subproblems, and conflict clauses between the clients. Like MiraXT, the *internal* communication among the threads of one client is carried out using the *Master Control Object*, while all *external* communication between the master process and the clients has been realized with MPICH [14], an implementation of the *Message Passing Interface* standard. From a top level point of view, PaMiraXT has some similarities to other approaches such as PSATO [30] or //Satz [15], which are based on a master/client model as well. The main difference is the fact, that in our approach each client itself is a parallel SAT algorithm, solving its current subproblem with a number of threads in parallel. Furthermore, the number of threads a client is using for SAT solving, can vary from client to client, making PaMiraXT perfectly adaptable to the workstation cluster under consideration. Again, the experimental results demonstrate that the runtime needed to solve a particular benchmark decreases significantly when using more clients.

The remainder of the article is structured as follows: in Section 2 we give an overview of the principles of SAT solving and address related work in the field of parallel SAT algorithms. Section 3 highlights the design and implementation of MiraXT, while PaMiraXT is covered in Section 4. Next, experimental results are discussed, followed by a conclusion and final remarks in Section 6.

2. SAT Solving & Related Work

In many different research areas, ranging from *Electronic Design Automation* [22] to *Artificial Intelligence* [17], problems can be described as a Boolean satisfiability problem and formatted in *Conjunctive Normal Form* (CNF). A CNF formula F consists of a conjunction of clauses, with each clause consisting of the inclusive disjunction of literals. A literal is the occurrence of a variable in its positive or negative phase. Finally, the Boolean satisfiability problem corresponds to the question, whether there exists an assignment for all variables occurring in F such that F evaluates to *true*.

2.1 Sequential SAT Solving

Based on the definitions given above, a sequential SAT solver starts with all variables of a CNF formula in an undefined state. Then, using a heuristic, a decision is made, assigning an undefined variable to *true* or *false*. Such a variable is called a *decision variable*. After every decision, a *Boolean Constraint Propagation* (BCP) procedure is run to find all implications resulting from that decision. Most solvers maintain a chronological list of decision variables and implications found by the BCP procedure. This list is usually sorted by increasing *decision levels* that are associated with every decision/implication, starting with the first decision and its implications on decision level 1. Decision level 0 however, contains implications that do not depend on a decision, e.g. implications forced by unit clauses. As the BCP procedure runs, it can also find conflicts, evoking a conflict analysis routine to find the reasons for the current conflict. This procedure would then try to resolve the conflict by backtracking to a previous decision level, undoing all the "wrong" variable assignments. It also records a conflict clause to prevent the conflict from being repeated. If the conflict analysis procedure detects a conflict on decision level 0, the problem is unsatisfiable because there is a conflict within the CNF formula without assigning at least one variable. Otherwise, selecting decision variables and running the BCP routine alternate until all variables have been assigned. In that case the problem is satisfiable and a satisfying variable assignment has been found. For a more in-depth overview of modern sequential SAT algorithms the reader is referred to [10].

2.2 Parallel SAT Solving

Typically, a parallel SAT solver consists of a collection of sequential SAT procedures, solving a problem in parallel. However, to obtain a fully functional parallel algorithm the sequential SAT routines have to be modified.

First, the overall CNF formula has to be divided into disjoint portions to be treated in parallel by the involved SAT solving processes. The most widely used way adopts a *Dynamic Search Space Partitioning* technique proposed by Zhang et al. in 1997 [30], which is based on the concept of *Guiding Paths* (GP). A GP is defined as a path in the search tree from the root (no variables assigned) to the current node (some variables assigned), with additional information attached to the edges:

1. The variable x_l and the truth value of x_l selected on decision level l.
2. A special flag storing the *type* of the variable x_l:
   - *Open* corresponds to a decision variable.
   - *Closed* corresponds to an implication.

Intuitively, a GP describes the current state of the search process by storing all assigned variables, their truth values, and the type of each variable. Due to the definition it is quite clear, that all open variables are potential candidates for dividing a CNF formula, because both truth values of every decision variable must be evaluated to prove unsatisfiability. So, if a SAT solving process is called to split its current subproblem by another (idle) process, it is in principle free to pick any open variable from its own GP to generate a new and unevaluated subproblem to be solved by an idle process. Not only in MiraXT and PaMiraXT, but in most parallel SAT solvers, the decision variable located closest to the beginning of the GP will be selected, corresponding to the shortest GP with respect to the number of "predefined" variables. The main motivation behind this strategy is that a short GP correlates to a large fraction of the overall search space, while a long GP corresponds to a smaller part that might be also easier and in particular faster to solve. Basically, the goal is to minimize the overall communication time by reducing the total number of such operations. Additionally, the selection of the decision variable located closest to the root also simplifies the encoding of the new and unchecked portion of the search space: all variables preceding the open variable can be taken as they are; only the truth value of the decision variable used as the *splitting point* has to be flipped. After sending the new GP to an idle process, the active SAT solving process finally marks the selected open variable to be *closed*. This ensures, that it will always ignore the specified GP as it will be analyzed by another process.

An example of how to use the guiding path method to divide a CNF formula is given in Figure 1. Besides the generation of new subproblems, the core routines of a parallel SAT algorithm have to be modified in the way, that they are able to start at any point within the search space specified by an initial guiding path.

Secondly, conflict clauses can be exchanged between parallel running SAT routines, allowing all processes to benefit from what each other has learnt. Each conflict clause deduced from a conflicting variable assignment is a piece of information that the corresponding process has learnt about the problem. Conflict clauses restrict the search of the solver by restricting the solution space. In a parallel context, the exchange of conflict clauses that directly force conflicts on other SAT solving processes are especially helpful in increasing the performance. In these cases the receiving process could immediately stop its analysis of the current part of the search space, since it has been already proven to be unsatisfiable by another process working on a different portion of the CNF formula. A side effect of this kind of *Knowledge Sharing* is the elimination of a complex termination mechanism, which would allow the processes to terminate other SAT solving units when they discover that these "areas" are superfluous for finding a satisfying assignment.

2.3 Related Work

Research on parallelizing SAT algorithms can be traced back to at least 1994, where Böhm and Speckenmeyer introduced an approach for a transputer system with up to 256 processors

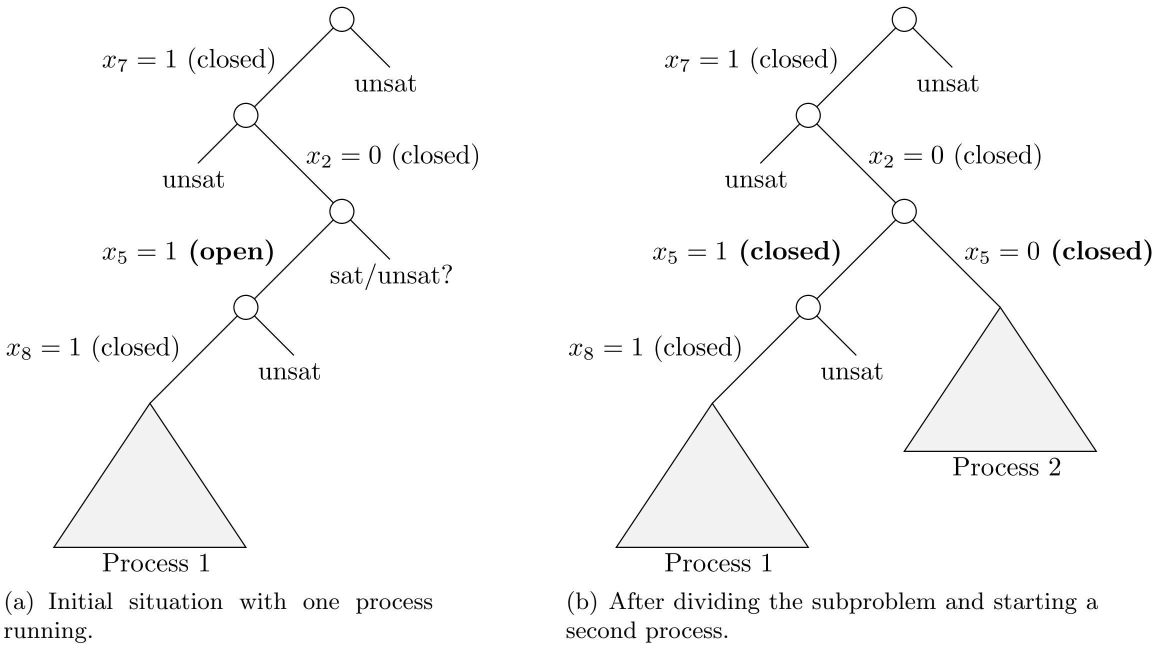

Figure 1. Example of dividing the search space. On the left-hand side, the initial situation with one process running is given. The current state of process 1 can be described by the guiding path $GP_1 = [(x_7, \mathit{closed}), (\neg x_2, \mathit{closed}), (x_5, \mathit{open}), (x_8, \mathit{closed})]$. On the right-hand side the situation after process 1 has split its subproblem and started a new process is sketched. Using GP_1 as the basis, the starting point of process 2 can be specified by the (initial) guiding path $GP_2 = [(x_7, \mathit{closed}), (\neg x_2, \mathit{closed}), (\neg x_5, \mathit{closed})]$. Finally, to guarantee that both processes work on disjoint parts of the overall problem, the guiding path of process 1 gets modified by marking the formerly open decision variable x_5 as *closed*: $GP_1' = [(x_7, \mathit{closed}), (\neg x_2, \mathit{closed}), (x_5, \mathit{closed}), (x_8, \mathit{closed})]$.

[5]. In subsequent years, a number of more advanced implementations have been published. To compare the various designs, we are going to focus on how the different solvers implement the exchange of conflict clauses. This is because clause sharing can significantly improve performance, but is usually also responsible for the vast majority of the communication. So, there is obviously a trade-off between the advantage of sharing conflict clauses to prune the search space and the communication overhead to broadcast this information, making knowledge sharing one of most critical points when designing a parallel SAT solver.

The first and most common way to support conflict clause sharing is by using message passing, typically according to the *Message Passing Interface* standard [27]. This concept has been used in various approaches [6, 15, 25, 30], all of them showing that significant speedup can be achieved. But due to the overhead associated with sending messages via some kind of network, typically only short clauses consisting of a very few literals are sent. Additionally, the clauses are often are sent in bundles, on one hand minimizing the communication overhead, but on the other hand introducing more latency into the clause sharing system. For example, in GridSAT [6] only clauses with at most three literals are exchanged between the different SAT solving processes, which is also the setup in PaMiraXT (on the client's side, see Section 4).

Another method maintains a shared *Conflict Clause Database* to share clauses, which has to be accompanied by a thread-based design. In this system, each thread occasionally sends *well-suited* clauses to this database (again, according to some criteria, e.g. the clause length), while also checking to see if new conflict clauses have been added by other SAT solving threads. PaSAT [26][1.] and ySAT [11] both use this concept, illustrations of the two approaches are given in Figures 2 and 3.

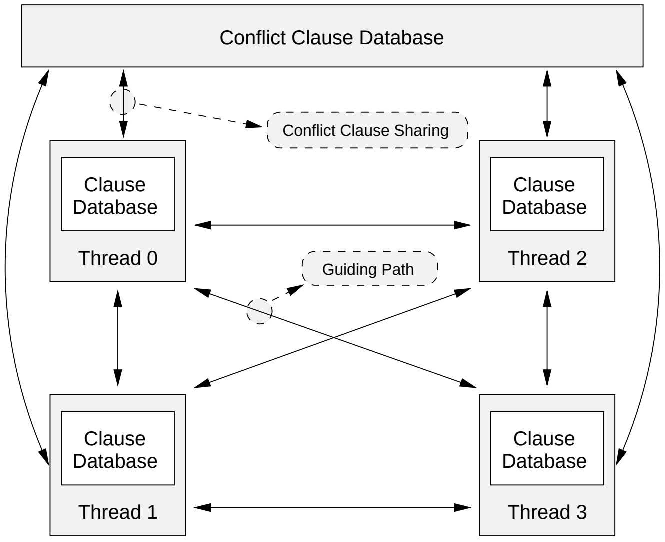

Figure 2. Block diagram of PaSAT

As can be seen by the figures, one of the main differences between PaSAT and ySAT is that in PaSAT every thread maintains its own (local) clause database, containing all clauses of the original CNF formula as well as all conflict clauses deduced by itself or copied from the *Conflict Clause Database*. In contrast to this, ySAT shares physical copies of each clause of the original CNF formula, while storing all conflict clauses locally within the private memory of the threads. Anyway, the concept of a shared *Conflict Clause Database* is typically not as scalable as message passing, but allows longer clauses to be shared without overwhelming the entire search process with communication related tasks. This system also reduces the latency within the clause sharing mechanism.

The third method is the shared memory clause database design that MiraXT (and by this the clients of PaMiraXT) use. Compared to ySAT, our approach goes one step further by maintaining only one physical copy of each clause, regardless of it is a conflict clause or a clause of the original CNF formula (see also Figure 4). All conflict clauses are added to this *Shared Clause Database*, and each SAT solving thread dynamically selects which clauses it wants to use. This is the reverse to the concept mentioned before, in which each thread chooses which clauses it wants to offer or send to the other threads by integrating them

1. Note, that different versions of PaSAT have been published in the past. Besides the one discussed here, there is also a version based on an *Algorithm Portfolio* design as well as a version using mobile agents when executed on a workstation cluster [3, 4].

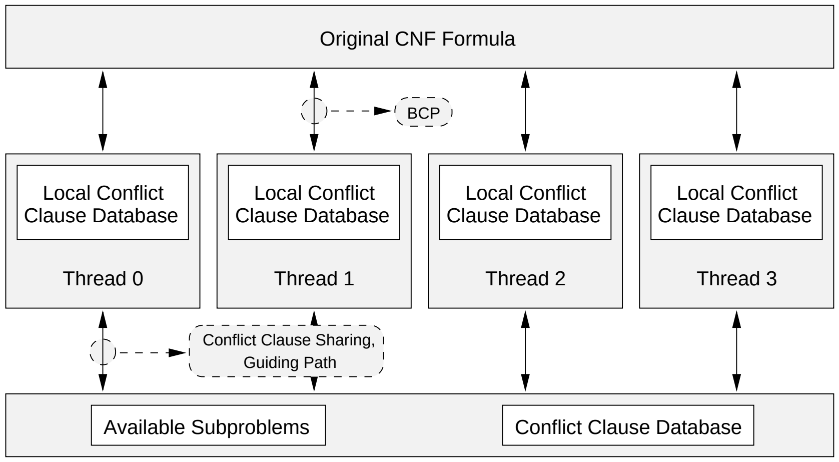
Original CNF Formula
BCP
Local Conflict Clause Database
Thread 0
Local Conflict Clause Database
Thread 1
Local Conflict Clause Database
Thread 2
Local Conflict Clause Database
Thread 3
Conflict Clause Sharing, Guiding Path
Available Subproblems
Conflict Clause Database

Figure 3. Block diagram of ySAT

into the *Conflict Clause Database*. As an example, in PaSAT only clauses with at most five literals will be shared. In contrast to this, our design allows each thread to check all available conflict clauses and to consider its current status when selecting helpful clauses. A thread can now decide to add very long clauses that will force implications or conflicts, while ignoring short clauses that are already fulfilled by the thread's variable assignments. This design takes full advantage of the low latency and high bandwidth a shared memory database provides.

Finally, with the introduction of pMiniSat and ManySAT two new parallel solvers have entered the special track on parallel SAT algorithms of this year's *SAT Race*. To our best knowledge only brief descriptions of the two approaches are currently available [12, 29], so it is not possible to provide a detailed analysis of either of them. Roughly speaking, pMiniSat is a parallel version of MiniSat (using message passing as the communication principle), while ManySAT is based on an algorithm portfolio design, running multiple different solvers in parallel, trying to capitalize on the fact that different solvers are better on different problems.

3. MiraXT

MiraXT is a zChaff-like SAT solver based on Mira [18, 19], but significantly improved and extended, so it can be run with multiple threads. The implementation was carried out using C++ and POSIX threads. As can be seen from the overall design given in Figure 4, MiraXT contains four main components: a preprocessing unit, the *Shared Clause Database*, the *Master Control Object*, and the SAT threads which are responsible for solving a CNF formula in parallel. In the following we discuss these components in more detail, leaving out the preprocessing unit, since it is more or less a re-implementation of what has been done in SatELite [9].

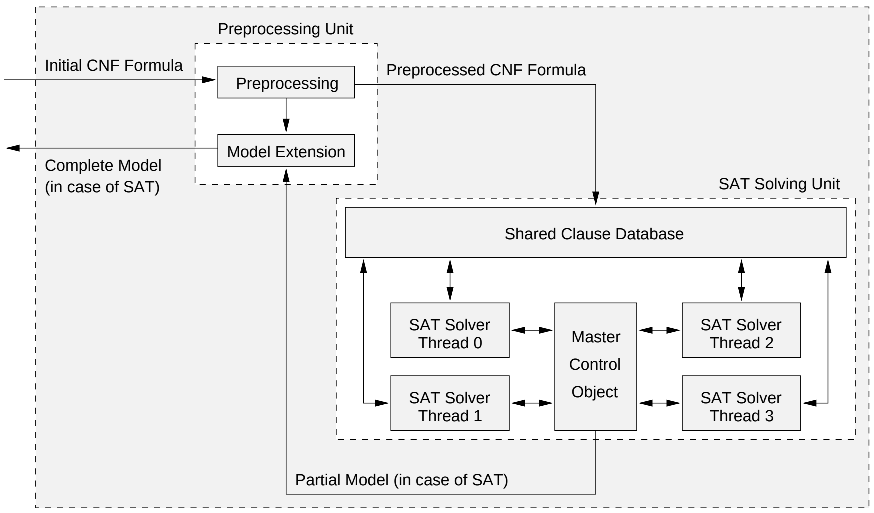

Figure 4. Block diagram of MiraXT

3.1 Shared Clause Database

As mentioned above, MiraXT has only one global clause database that stores all clauses of the original CNF formula, plus all conflict clauses deduced by the threads during runtime. Figure 5 gives an overview of the design of the *Shared Clause Database*. It mainly consists of pointers to clauses, while the clauses themselves consist of the literals of the clause (in the figure denoted by *Clause*), the clause length (|C|) and the so-called *Clause Deletion Flags* (represented by small boxes at the end of each clause, in this example assuming that four threads are running in parallel). These Boolean flags indicate, whether a SAT thread is currently using a particular clause of the *Shared Clause Database* to solve its subproblem or not.

In order to insure coherency within the database, a mutex lock must be acquired before a thread inserts a pointer to a conflict clause (deduced by its conflict analysis routine) into the *Shared Clause Database*. As soon as the pointer is inserted and the database clause counter is incremented (two simple operations) the lock is released. All clauses, once added to the clause database, are read-only, so that sharing can be done without locks. These steps are important as we want to reduce the amount of locks needed by the solver, thus minimizing the waiting times of the threads that might occur due to lock contentions.

Furthermore, each thread has a pointer associated with it that is used when it requests new clauses from the *Shared Clause Database*. This pointer keeps track of which clauses the thread has already looked at, and those that are new. In Figure 5 this special pointer is plotted for thread 0, but it could represent other threads as well.

Clause deletion is also an important feature of modern SAT engines. In MiraXT, each thread deletes clauses using an algorithm similar to BerkMin [13] in which older and more inactive clauses are easier to delete. To facilitate clause deletion efficiently in our multi-threaded design, the threads use the *Clause Deletion Flags* associated with each clause. Whenever a thread deletes its reference to a clause, it sets its clause deletion flag for that particular clause. Because this Boolean flag is specific for that thread, no lock is required

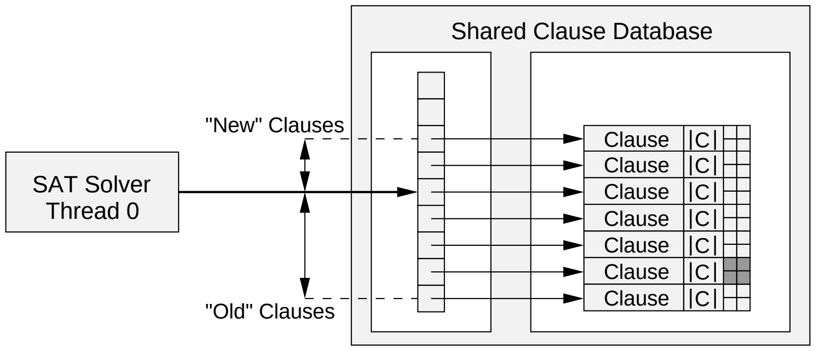

Figure 5. Shared Clause Database

when deleting clauses. Once a thread has deleted all the clauses it wants to delete, it will ask the shared clause database to see if a global clause deletion procedure should be run, as the threads only remove their references to clauses, and not the actual clauses themselves. If the database needs cleaning, the thread locks it and proceeds to physically delete clauses that are no longer used by any thread, relinquishing the lock when it is finished. This lock is used to insure that no two threads run the global clause deletion procedure at the same time. However, during this time all other threads can continue running normally. Assuming the situation given in Figure 5, the clause having all its *Clause Deletion Flags* plotted in dark grey can be removed, since the flags state that none of the four threads is using this particular clause any more.

Using the fine grained lock system described above, practically all lock contention issues were removed, and in testing we saw no signs of even light lock contention. This seemed to be one of the problems of ySAT [11], as the authors report that with four threads, an average of 10% of the time was spent waiting for locks. This number should be only fractions of a percent, as is indeed the case for MiraXT on most problems. This means that in MiraXT, the majority of the time (over 99% on average) is spent actually solving the problem.

3.2 Master Control Object

MiraXT contains no separate master process unlike parallel solvers based on a classical master/client model. Instead a *Master Control Object* (MCO) allows the SAT solving threads to communicate with each other. All communication is done in a passive way, such that the MCO will not interfere with the threads. It will only store messages and suspend threads which ask for it to do so. The SAT solving threads poll the MCO occasionally to see if there any messages or idle threads, waiting for a new subproblem to evaluate.

In principle, MiraXT's threads and the *Master Control Object* interact as follows: after preprocessing the CNF formula thread 0 starts solving the entire problem. All other threads start by requesting a subproblem from the MCO and are now waiting to be signaled. Idle threads are not wasting CPU cycles, polling for a new subproblem, they are put to sleep and awakened using the POSIX `cond_wait` / `cond_signal` commands instead. Periodically, running threads ask the MCO for any new global events (e.g. problem solved, waiting threads, time limit exceeded), which can be done without using locks. If something has happened, a more complicated procedure with a mutex lock will be executed. In our example, when thread 0 checks the MCO, it will recognize that other threads are waiting for a subproblem. It will then ask the MCO for the mutex lock associated with the MCO. Once thread 0 has acquired this lock, it will split its current subproblem as described in Section 2.2, and add the new subproblem (encoded as a guiding path) to the MCO. Afterwards, thread 0 will awake a sleeping thread, release the MCO lock, and finally continue solving its part of the problem.

If there are more than just one thread waiting, they will be served randomly by running threads. No heuristic is used to decide which running thread should split its current subproblem. Here, the goal is to minimize the communication overhead and by this to speedup the SAT solving process. If a thread proves its subproblem to be unsatisfiable, it will request a new subproblem. If all SAT solving threads involved in the search process are waiting for a new subproblem, the CNF formula is unsatisfiable. Otherwise, if any thread finds a solution, the entire problem is satisfiable.

3.3 SAT Threads

The SAT solving threads used in MiraXT are based on the classical Davis-Putnam method, but enhanced with the features of a modern SAT engine such as conflict-driven learning with non-chronological backtracking, watched literals for lazy clause evaluation, restarts, clause deletion, and an efficient decision heuristic, combining the VSIDS strategy known from zChaff with RSat's concept of *Assignment Caching* [24]. In the following we highlight the modifications we have done with respect to *Boolean Constraint Propagation* and the analysis of conflicts, both strongly depending on the architecture of the *Shared Clause Database* discussed in Section 3.1.

In most solvers, to keep track of the watched literals, the original clause is modified in some way, e.g. by using the first two literals in the clause. This is not possible in MiraXT, because clauses are used by many threads and because of that are read-only after being inserted. So, each thread creates a second data structure called the *Watched Literals Reference List* (WLRL) to handle its watched literals.

As can be seen from Figure 6, the WLRL not only stores the two watched literals (WL) for each clause, but also a pointer to the entire clause and – depending on the clause length – up to two additional literals, the so-called *Cache Literals* (CL).[2.] The CLs are the ones that get checked first, if one of the two watched literals of a clause has to be updated. If none of the two CLs can serve as the new watched literal, then the entire clause has to be evaluated literal by literal (in case the corresponding clause consists of five or more literals, otherwise the clause is completely stored within the WLRL). Intuitively, the *Watched Literals Reference List* allows each thread to have a condensed copy of every clause. In [20] it has been shown empirically, that updating the watched literals of "long" clauses not completely stored in the WLRL can be done by evaluating the CLs only in about 84% of all cases. This means the original clauses are not needed 84% of the time, making MiraXT more cache friendly.

The conflict analysis procedure in MiraXT is based on the *1UIP* scheme known from zChaff [31]. However, a separate clause addition procedure was added. In MiraXT, the conflict analysis procedure of a thread will first add a clause to the shared clause database. Then the clause addition procedure will be run, asking the clause database for all new

2. Like in most modern SAT solvers, unit clauses and binary clauses are handled separately in MiraXT.

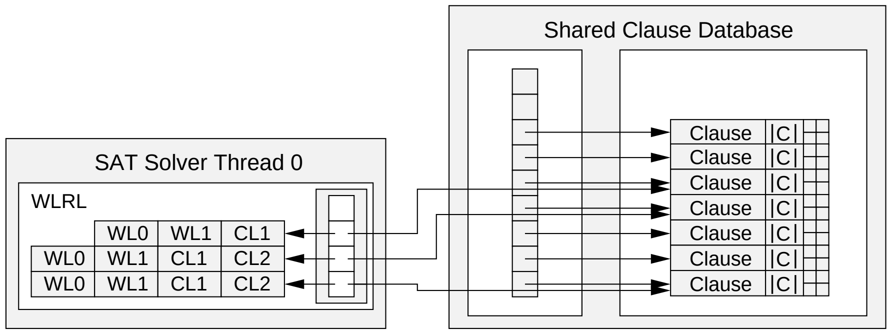

Figure 6. Watched Literals Reference List

clauses; this includes clauses generated by other threads and its newly generated conflict clause. It will then process these clauses, deciding which clauses should be considered in its own search process. Currently, all clauses directly leading to a conflict, clauses forcing an implication, and really short clauses (10 literals or less), are added to the thread's WLRL. The clause addition procedure will set the watched literals, search for implications, and perform conflict-driven backtracking as needed. Both the conflict analysis procedure and the clause addition procedure can signal that the current subproblem is unsatisfiable.

4. PaMiraXT

In this section we describe the main properties of PaMiraXT, a development that has been carried out to handle multiple instances of MiraXT in parallel. A block diagram of PaMiraXT is sketched in Figure 7. As can be seen, the overall design follows a master/client model, in which one process is dedicated to be the master, while a number of copies of MiraXT act as clients. In more detail the role of the master is...

– to start/stop the clients,

– to evaluate the workload of all running clients if one copy of MiraXT becomes idle during runtime and needs a new subproblem,

– to select the *most loaded* client and to request an unevaluated part of the search space from it,

– to forward this problem to the idle client, and

– to receive and broadcast conflict clauses from/to the clients.

Likewise the role of the clients is...

– to receive and solve a subproblem,

– to return an unevaluated fraction of the entire formula when asked by the master,

– to receive and forward *well-suited* conflict clauses from/to the master process, and

– to send the results to the master.

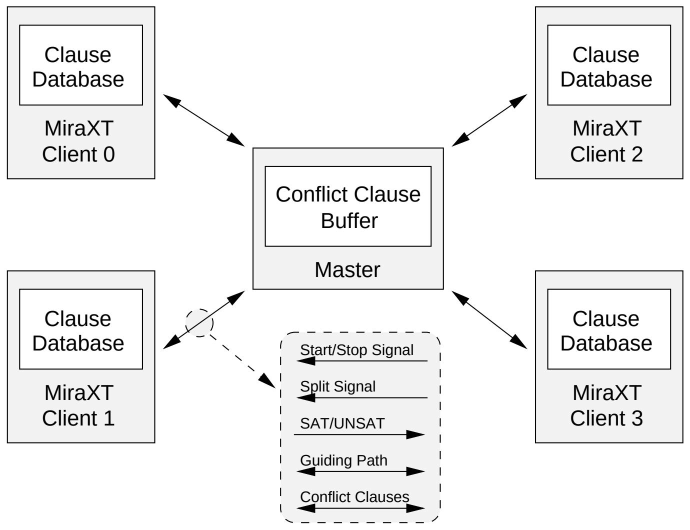

Figure 7. Block diagram of PaMiraXT

As the underlying communication protocol the MPICH implementation [14] of the *Message Passing Interface* standard was chosen, providing a standardized core of C/C++ library routines to easily develop distributed algorithms. All communication tasks are sent as messages using the (blocking) send operation `MPI_Send`. Since the master is active only when receiving and processing requests from the clients (and stays idle in all other cases), it will be temporarily suspended by the `MPI_Probe` command. By doing so, we are able to execute the master on the same workstations as the clients without reducing the performance significantly. Of course, one has to set the parameters for exchanging conflict clauses between the clients in such a way, that the effort of the master process to broadcast this information does not slow down the client running on the same workstation (see also Section 4.2).

4.1 Dynamic Search Space Partitioning

Based on the concept of *Guiding Paths* as introduced in Section 2.2, the exchange of subproblems between the clients of PaMiraXT is done as follows: If one client becomes idle during runtime it sends a request to the master process. The master then has to determine which of the running copies of MiraXT should be chosen to provide a new unchecked portion of the search space. Usually, there is more than one process active, so it is up to the master to decide which one should be selected. Again, as mentioned before, motivated by the fact, that a short GP (with respect to the number of literals) correlates to a large fraction of the search tree, while a long GP indicates a smaller part (that might be also easier and faster to solve), the master always calls the client with the shortest guiding path to divide its subproblem.

Similar to MiraXT, during the initialization phase, only one client is started by the master process by giving it an empty guiding path (equivalent to the overall problem), while all other clients remain idle. As a consequence the "work stealing" mechanism presented before is called immediately by all remaining idle clients.

4.2 Knowledge Sharing

The exchange of conflict clauses among the clients involved in the search process has been put into practice according to three phases: selecting suitable clauses by the clients, filtering the received information on the master's side, and finally integrating the shared conflict clauses into every client's database, so that they are available for all threads.

In the current version of PaMiraXT all conflict clauses within the *Shared Clause Database* of a client that don't exceed a length of three literals will be selected for sharing. To reduce the communication overhead, each client uses its own *Conflict Clause Buffer* to transfer several conflict clauses as one single MPI message to the master process. All conflict clauses with at most three literals will be stored in this buffer, and communication takes place only when it is completely filled. In the experiments we used a buffer size of 50, which turned out to be a good compromise between the communication effort and the number and the "quality" of the conflict clauses being transferred.

In the second phase, the master does not simply broadcast the received information to all clients, but performs a *filtering step* first to check which of the initial guiding paths of the clients already satisfy the received clauses.[3.] Finally, a clause is transferred only to those clients where it is not subsumed by the corresponding initial GP. Similar to the first phase, the master also uses a separate outgoing buffer for each client (specified by the same parameters as above) to forward a set of clauses per communication task. The idea of discarding already satisfied clauses is that these pieces of information neither force an implication, nor are they responsible for a conflicting assignment in the actual part of the search tree. So, they are currently not helpful to guide the algorithm more efficiently, but reduce the overall performance due to the communication time and the increased number of clauses the BCP routines of the clients probably have to deal with.

Finally, the receiving clients take all conflict clauses sent as one MPI message and integrate them into their own *Shared Clause Database*, so that they are directly available for the SAT solving threads of that particular client.

4.3 Clients

To use MiraXT as a client of PaMiraXT in the way described in the previous sections, we extended the design by introducing an additional thread, the so-called *MPI thread*. An example of such a modified version of MiraXT, containing four SAT solving threads as well as one MPI thread is represented in Figure 8.

The MPI thread is not participating in the search process, but is only responsible for the overall communication with the master process. Like the SAT threads, it has access to the *Master Control Object*, mainly used for dividing the search space and sending/receiving subproblems to/from the master. If a client is called to divide its current subproblem by the master process, the corresponding MPI thread behaves like the SAT threads in such a situation: it requests a new subproblem within its *Master Control Object* and waits until one of the running SAT threads has added a guiding path, specifying a new subproblem. The MPI thread will then pick this guiding path and transfers it to the master process, so that the master is able to serve an idle client. On the side of the idle client, the corresponding

3. Due to the fact, that the master is also responsible for exchanging subproblems, it always knows the initial GPs of all clients.

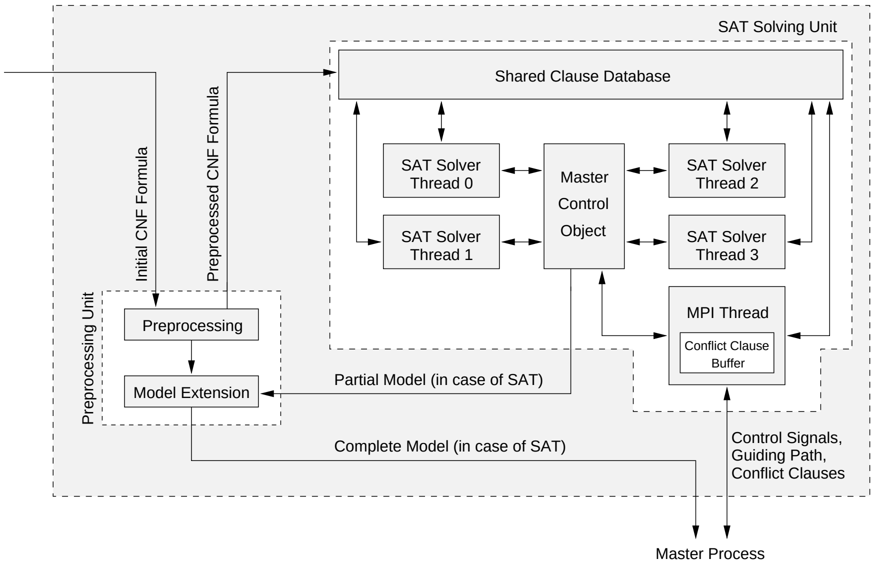

Figure 8. Block diagram of a PaMiraXT client

MPI thread will receive the subproblem and adds it to the MCO, where it gets solved by the SAT threads in parallel.[4.]

As can be seen from Figure 8, the MPI thread also has an interface to the *Shared Clause Database*, which is used to exchange conflict clauses between the clients of PaMiraXT. For this purpose, the MPI thread periodically checks which of the conflict clauses of the database might be helpful to other clients. It then copies such clauses into its internal *Conflict Clause Buffer* and submits the entire buffer to the master process if its completely filled. On the other side, if the MPI thread receives a bundle of clauses from the master, it adds all the clauses to its shared clause database, so that the SAT threads can immediately benefit from the information learnt by other clients, that are running on other workstations.

Finally, each client of PaMiraXT preprocesses the CNF formula on its own. Since they perform deterministic operations only, all clients end up with the same preprocessed formula. Compared to an approach doing the preprocessing step on the master's side and broadcasting the preprocessed formula to all clients afterwards, our strategy is less time-consuming.

5. Experimental Results

To measure the performance of our approach we conducted a large number of experiments, comparing MiraXT/PaMiraXT with other state-of-the-art SAT engines, i.e. MiniSat2, PicoSAT 535, and RSat 2.01. All comparisons were executed on an AMD workstation with two *Dual-Core AMD Opteron 280* processors, allowing MiraXT – the same as PaMiraXT with just one client – to be run with up to four threads (see Figure 9).

4. Note, that an idle client means, that all SAT threads of that client are waiting for a new subproblem. As soon as the MPI thread submits a new subproblem to the MCO, this subproblem will be immediately divided again and again, until all SAT threads are running.

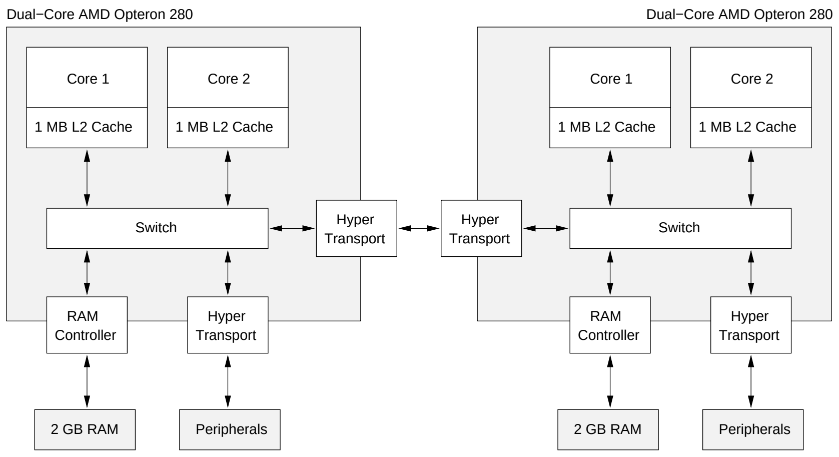

Figure 9. AMD system with two *Dual-Core AMD Opteron 280* processors [1]

To evaluate the potential of PaMiraXT we combined this workstation with another two AMD systems, which are almost identical to the one sketched in Figure 9. The only difference is that these two workstations are equipped with two single-core *AMD Opteron 250* processors each. All workstations are running at 2.4 GHz clock frequency under a Linux SMP enabled kernel (version 2.6.24), and have 4 GB of main memory. With these three workstations in parallel, we have a (small) cluster with a total of 6 CPUs and 8 processor cores, capable for executing PaMiraXT with 3 clients and 8 SAT threads (1×4 threads and 2×2 threads).

A total of 347 instances were selected, containing all instances from the *SAT Race 2006* as well as the industrial benchmarks from the *SAT Competition 2007* [28]:

– *SAT Race 2006* – a mix of 100 satisfiable and unsatisfiable CNF formulas. Like in the original competition, we set the time limit to 900 seconds per benchmark.

– *SAT 2007 Industrial* – a mix of 247 satisfiable and unsatisfiable benchmarks, all of them coming from industrial applications. According to the setup used during the second stage of the competition, we defined a time limit of 10000 seconds per benchmark.

Due to the inherent non-determinism of parallel SAT solvers we ran MiraXT/PaMiraXT three times on all instances of the *SAT Race 2006* benchmark suite.[5.] The number of solved instances as well as the runtimes given in subsequent tables always represent the average of these three individual runs. Due to the high CPU limit chosen for the *SAT 2007 Industrial* problems, we executed our solvers only once on this particular set of CNF formulas.

In the following we discuss the results listed in Tables 1 to 5. As can be seen in Tables 1 and 2, even in the single-threaded mode, MiraXT compares well to the other sequential SAT algorithms, solving more instances within the given CPU limit, while being also significantly faster. On the *SAT Race 2006* benchmark class, MiraXT solves an additional instance

5. The overall search process strongly depends on conflict clause sharing and search space partitioning. The efficiency of these methods is highly influenced by the workload of the CPUs and the status of the communication channels, which might change from one run to another.

Table 1. Runtime & solved instances (SAT Race 2006, CPU limit: 900s)

SAT algorithm	Overall runtime [s]	Solved instances
RSat	40209.10	68
MiniSat2	41631.15	70
PicoSAT	37033.29	77
MiraXT, 1 Thread	32264.09	78
MiraXT, 2 Threads	27651.93	81
MiraXT, 4 Threads	21674.50	85
PaMiraXT, 3 Clients, 8 Threads	**21274.41**	**86**

Table 2. Runtime & solved instances (SAT 2007 Industrial, CPU limit: 10000s)

SAT algorithm	Overall runtime [s]	Solved instances
RSat	1293297.72	136
MiniSat2	1199874.43	143
PicoSAT	1110696.47	152
MiraXT, 1 Thread	999154.42	162
MiraXT, 2 Threads	835851.65	177
MiraXT, 4 Threads	783190.38	180
PaMiraXT, 3 Clients, 8 Threads	**711278.80**	**186**

within 900 seconds compared to PicoSAT, while being about 15% faster. The same holds for the *SAT 2007 Industrial* benchmarks. Again, even the single-threaded version of MiraXT is faster than MiniSat2, RSat, and PicoSAT and solves significantly more instances.

When switching from one to four threads running in parallel, the performance of MiraXT increases on both benchmark classes, resulting in a runtime reduction of 71% compared to PicoSAT (*SAT Race 2006*; 42% on the *SAT 2007 Industrial* benchmarks) and a total of 85 instances solved within the time limit (*SAT Race 2006*; 180 on the *SAT 2007 Industrial* benchmarks).

Furthermore, compared to the configuration of MiraXT with four threads running in parallel, Tables 1 and 2 clearly show, that PaMiraXT with three clients and eight threads is able to solve even more instances in less run time. Especially the results for the *SAT 2007 Industrial* instances are impressive, where PaMiraXT was able to solve another six hard CNF formulas! On the other hand, the results for the benchmark class *SAT Race 2006* also show, that the performance gain obtained by PaMiraXT is only marginal. As the main reason we identified that the majority of the instances of this benchmark suite can be solved very fast (in less than 100 seconds), making it hard to achieve a significant benefit when running more and more threads in parallel. Basically, the overhead of initializing and synchronizing the clients/threads does not pay off, if a benchmark can be "easily" solved. This also has an effect on the speedup as can be seen in Tables 3 and 4. While MiraXT/PaMiraXT scale well on the *SAT 2007 Industrial* benchmarks, the speedup con-

Table 3. Speedup of MiraXT/PaMiraXT (SAT Race 2006, CPU limit: 900s). To obtain correct speedup values only instances solved by all configurations of MiraXT and PaMiraXT were taken into account.

SAT algorithm	Overall runtime [s]	Speedup
MiraXT, 1 Thread	11353.62	1.00
MiraXT, 2 Threads	6902.29	1.64
MiraXT, 4 Threads	4418.97	2.57
PaMiraXT, 3 Clients, 8 Threads	4332.08	2.62

Table 4. Speedup of MiraXT/PaMiraXT (SAT 2007 Industrial, CPU limit: 10000s). To obtain correct speedup values only instances solved by all configurations of MiraXT and PaMiraXT were taken into account.

SAT algorithm	Overall runtime [s]	Speedup
MiraXT, 1 Thread	132618.80	1.00
MiraXT, 2 Threads	53734.05	2.47
MiraXT, 4 Threads	37769.97	3.51
PaMiraXT, 3 Clients, 8 Threads	23584.20	5.62

Table 5. Scaling of MiraXT in terms of deduced conflict clauses per second (SAT Race 2006, CPU limit: 900s)

No. of threads	No. of conflict clauses deduced per second
1 Thread	4149.88
2 Threads	7136.50
4 Threads	13118.85

verges to about 2.6 for the *SAT Race 2006* benchmark suite, regardless of whether four or eight threads are used in parallel.

As mentioned in Section 3.1, first experiments with MiraXT indicated, that the lock system used to ensure data consistency of the shared clause database forces only very few lock contentions during runtime. Table 5 underlines this fact by measuring the conflict clauses deduced and integrated into the database per second. Even with four threads the number of clauses added to the *Shared Clause Database* per second increases remarkable with the number of threads.

In the near future, we currently foresee two main problems with scaling MiraXT (either in the "stand-alone" version or as a client of PaMiraXT) to larger multi-core designs, e.g. 16 or 32 cores. The first one is the limited bandwidth of the front side bus which connects the CPU to main memory. Large industrial benchmarks running with multiple threads can require 1–50 GB of active memory. With 32 threads, the bandwidth requirements that are expected from this bus are significantly higher than just for one thread as no problem fits entirely in cache. Memory bandwidth is a problem for many high performance computing applications. This means that most hardware companies are putting significant effort into alleviating these problems. Some solutions such as widening the front side bus from 1–2 channels to 3–4 channels have already been done. Larger on chip caches are also being introduced. Both AMD and Intel now have L3 caches included on all their server CPUs. Lastly, cache performance techniques such as our WLRL and others (for example the ones integrated in [12]) will become increasingly more important.

The second problem is a bit more algorithmic. In MiraXT, each thread can examine all the clauses in the clause database. However, if we now have 32 threads, the number of conflict clauses produced is enormous, and each SAT solving thread will suffer from a form of information overload. Instead of executing MiraXT with 32 threads, we can solve this problem by running PaMiraXT with four clients, each running eight SAT threads (or two clients with 16 threads each). In this way, we can have multiple groups of tightly coupled threads that work together sharing everything, while also limiting the sharing between groups so that the number of conflict clauses each thread examines is more manageable. In the future, using ideas such as thread affinity, we can then specify which threads should run on which processor. This will allow us to generate rules that dictate which threads in MiraXT should work together. For example, we could allow threads that are connected to the same L2 or L3 cache to share more clauses then threads working on processors that are one or two *Hyper Transport* hops away from each other (see Figure 9).

Now, while we do expect new issues to crop up when adapting our ideas for larger future systems with more cores, we see no road blocks that should prevent our algorithms from scaling in the near future (16–32 cores). After 32 threads, the parallel performance gains are less clear, but so is the architecture of the CPUs and the platforms the solvers will be running on.

6. Conclusion

As was shown in this article, PaMiraXT is a powerful parallel SAT solver, providing significant speedup when executed with multiple threads/clients. The realization follows a master/client model, using copies of MiraXT as clients controlled by a separate master process. Concerning the number of threads, PaMiraXT can be perfectly adapted to the hardware environment under consideration by varying the number of SAT solving threads from client to client.

Parallel approaches will likely be the way of the future in the field of SAT solving as many chip manufacturers are turning from single-core to multi-core processors. Utilizing the extra power of these CPUs, regardless of a single workstation or a cluster of workstations is used, will be a fundamental issue in the future.

References

[1] AMD, Inc. *Next Generation AMD Opteron Processor with Direct Connect Architecture: 2P Server and Workstation Comparison*, 2006.

[2] F. Bacchus and J. Winter. Effective Preprocessing with Hyper-Resolution and Equality Reduction. In *6th International Conference on Theory and Applications of Satisfiability Testing*, pages 341–355, 2003.

[3] W. Blochinger, C. Sinz, and W. Küchlin. A Universal Parallel SAT Checking Kernel. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1720–1725, 2003.

[4] W. Blochinger, C. Sinz, and W. Küchlin. Parallel Propositional Satisfiability Checking with Distributed Dynamic Learning. *Parallel Computing*, **29**(7):969–994, 2003.

[5] M. Böhm and E. Speckenmeyer. A Fast Parallel SAT-Solver – Efficient Workload Balancing. *Annals of Mathematics and Artificial Intelligence*, **17**(3–4):381–400, 1996.

[6] W. Chrabakh and R. Wolski. GridSAT: A Chaff-based Distributed SAT Solver for the Grid. In *ACM/IEEE Supercomputing Conference*, 2003.

[7] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM*, **5**:394–397, 1962.

[8] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, **7**(3):201–215, 1960.

[9] N. Eén and A. Biere. Effective Preprocessing in SAT through Variable and Clause Elimination. In *8th International Conference on Theory and Applications of Satisfiability Testing*, pages 61–75, 2005.

[10] N. Eén and N. Sörensson. An Extensible SAT-Solver. In *6th International Conference on Theory and Applications of Satisfiability Testing*, pages 502–518, 2003.

[11] Y. Feldman, N. Dershowitz, and Z. Hanna. Parallel Multithreaded Satisfiability Solver: Design and Implementation. *Electronic Notes in Theoretical Computer Science*, **128**(3):75–90, 2005.

[12] G. Chu, P. Stuckey, and A. Harwood. *PMiniSat Solver Description*, 2008.

[13] E. Goldberg and Y. Novikov. BerkMin: A Fast and Robust SAT-Solver. In *Design, Automation, and Test in Europe*, pages 142–149, 2002.

[14] W. Gropp, E.L. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, **22**(6):789–828, 1996.

[15] B. Jurkowiak, C.M. Li, and G. Utard. Parallelizing Satz using Dynamic Workload Balancing. In *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing*, pages 205–211, 2001.

[16] H.A. Kautz, E. Horvitz, Y. Ruan, C. Gomes, and B. Selman. Dynamic Restart Policies. In *18th AAAI National Conference on Artificial Intelligence*, pages 674–681, 2002.

[17] H.A. Kautz and B. Selman. Planning as Satisfiability. In *10th European Conference on Artificial Intelligence*, pages 359–363, 1992.

[18] M. Lewis, T. Schubert, and B. Becker. Early Conflict Detection Based BCP for SAT Solving. In *7th International Conference on Theory and Applications of Satisfiability Testing*, pages 29–36, 2004.

[19] M. Lewis, T. Schubert, and B. Becker. Speedup Techniques Utilized in Modern SAT Solvers – An Analysis in the MIRA Environment. In *8th International Conference on Theory and Applications of Satisfiability Testing*, pages 437–443, 2005.

[20] M. Lewis, T. Schubert, and B. Becker. Multithreaded SAT Solving. In *12th Asia and South Pacific Design Automation Conference*, pages 926–931, 2007.

[21] J.P. Marques-Silva and K.A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, **48**(5):506–521, 1999.

[22] J.P. Marques-Silva and K.A. Sakallah. Boolean Satisfiability in Electronic Design Automation. In *IEEE/ACM Design Automation Conference*, pages 675–680, 2000.

[23] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *IEEE/ACM Design Automation Conference*, pages 530–535, 2001.

[24] K. Pipatsrisawat and A. Darwiche. A Lightweight Component Caching Scheme for Satisfiability Solvers. In *10th International Conference on Theory and Applications of Satisfiability Testing*, pages 294–299, 2007.

[25] T. Schubert, M. Lewis, and B. Becker. PaMira – a Parallel SAT Solver with Knowledge Sharing. In *6th International Workshop on Microprocessor Test and Verification*, pages 29–34, 2005.

[26] C. Sinz, W. Blochinger, and W. Küchlin. PaSAT – Parallel SAT-Checking with Lemma Exchange: Implementation and Applications. In *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing*, 2001.

[27] M. Snir, S.W. Otto, D.W. Walker, J. Dongarra, and S. Huss-Lederman. *MPI: The Complete Reference*. MIT Press, 1995.

[28] The International SAT Competitions Web Page. `http://www.satcompetition.org/`.

[29] Y. Hamadi, S. Jabbour, and L. Sais. *ManySAT Solver Description*, 2008.

[30] H. Zhang, M. Bonacina, and J. Hsiang. PSATO: a Distributed Propositional Prover and its Application to Quasigroup Problems. *Journal of Symbolic Computation*, **21**(4):543–560, 1996.

[31] L. Zhang, C.F. Madigan, M.W. Moskewicz, and S. Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 279–285, 2001.

[32] L. Zhang and S. Malik. Cache Performance of SAT Solvers: a Case Study for Efficient Implementation of Algorithms. In *6th International Conference on Theory and Applications of Satisfiability Testing*, pages 287–298, 2003.